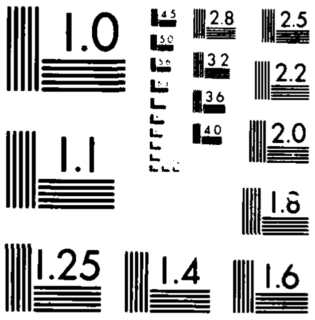
AD-A164 897 SEMI-APPLICATIVE PROGRAMMING EXAMPLES OF CONTEXT FREE RECOGNIZERS(U) BBN LABS INC CAMBRIDGE MA N S SRIDHARAN JAN 86 BBN-6135 N00014-77-C-0378 1/1

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



(12)

BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 6135

Semi-Applicative Programming: Examples of Context Free Recognizers

N.S. Sridharan

January 1986

Prepared for:
Defense Advanced Research Projects Agency

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER BBN Report No. 6135	2. GOVT ACCESSION NO. AD-A164 897	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SEMI-APPLICATIVE PROGRAMMING: EXAMPLES OF CONTEXT FREE RECOGNIZERS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER BBN Report No. 6135
7. AUTHOR(s) N. S. Sridharan		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0378 N00014-85-C-0079
9. PERFORMING ORGANIZATION NAME AND ADDRESS BBN Laboratories Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy Arlington, VA 22217		12. REPORT DATE January 1986
		13. NUMBER OF PAGES 41
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence, programming language, parallelism, semi-applicative programming, context-free parsing, Earley Algorithm, Cocke-Kasami-Younger Algorithm, program transformation, program annotation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Most current parallel programming languages are designed with a sequential programming language as the base language and have added constructs that allow parallel execution. We are experimenting with an applicative base language that has *implicit* parallelism everywhere, and then we introduce constructs that *inhibit* parallelism. The base language uses pure LISP as a foundation and blends in interesting features of Prolog and FP. Proper utilization of available machine

20.

resources is a crucial concern of programmers. We advocate several techniques of controlling the behavior of functional programs without changing their meaning or functionality: program annotation with constructs that have benign side-effects, program transformation and adaptive scheduling. This combination yields us a *semi-applicative* programming language and an interesting programming methodology.

In this paper we deal with context-free parsing as an illustration of semi-applicative programming. Starting with the specification of a context-free recognizer, we have been successful in deriving variants of the recognition algorithm of Cocke-Kasami-Younger. One version is the CKY algorithm in parallel. The second version includes a top-down predictor to limit the work done by the bottom-up recognizer. The third version uses a cost measure over derivations and produces minimal cost parses using a dynamic programming technique. In another line of development, we arrive at a parallel version of the Earley algorithm. All of these algorithms reveal more concurrency than was apparent at first glance.

Semi-Applicative Programming: Examples of Context Free Recognizers

N.S. Sridharan

AI Department, BBN Laboratories Inc.

10 Moulton St, Cambridge, MA 02238

Arpanet: Sridharan@BBNG

31 January 1986

This research was sponsored by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. N00014-85-C-0079. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government

Table of Contents

Abstract 1
1. Semi-Applicative Programming 2
  1.1. Motivation 3
  1.2. Experience 3
2. Detailed Discussion of our Approach 5
  2.1. Computation Model 6
  2.2. Base language: SALT 6
  2.3. Resource control in applicative programs 8
    2.3.1. Program annotations 8
    2.3.2. Program transformation 8
    2.3.3. Adaptive scheduling 9
  2.4. Annotation language: PEPPER 9
3. Research Results: Derivation of recognition algorithms for CNF grammars 11
  3.1. Preliminaries: Grammar, Normal Form and Derivation 11
  3.2. Derivation of a parallel top-down recognizer 14
  3.3. Derivation of a parallel bottom-up recognizer 16
  3.4. Derivation of a dynamic programming technique 21
  3.5. Derivation of a predictive recognizer 23
    3.5.1. Weakening the predictor 27
    3.5.2. Relaxed synchronization of predictor and recognizer 29
    3.5.3. Adaptive scheduling of tasks 30
  3.6. Derivation of a version of Earley's algorithm 30
4. Conclusion 36
Postscript 38
Acknowledgement 38

Abstract

Most current parallel programming languages are designed with a sequential programming language as the base language and have added constructs that allow parallel execution. We are experimenting with an applicative base language that has *implicit* parallelism everywhere, and then we introduce constructs that *inhibit* parallelism. The base language uses pure LISP as a foundation and blends in interesting features of Prolog and FP. Proper utilization of available machine resources is a crucial concern of programmers. We advocate several techniques of controlling the behavior of functional programs without changing their meaning or functionality: program annotation with constructs that have benign side-effects, program transformation and adaptive scheduling. This combination yields us a *semi-applicative* programming language and an interesting programming methodology.

~~In~~ this paper we deal with context-free parsing as an illustration of semi-applicative programming. Starting with the specification of a context-free recognizer, we have been successful in deriving variants of the recognition algorithm of Cocke-Kasami-Younger. One version is the CKY algorithm in parallel. The second version includes a top-down predictor to limit the work done by the bottom-up recognizer. The third version uses a cost measure over derivations and produces minimal cost parses using a dynamic programming technique. In another line of development, we arrive at a parallel version of the Earley algorithm. All of these algorithms reveal more concurrency than was apparent at first glance.

1. Semi-Applicative Programming

Novel parallel machines are being designed and built. Amid the loud applause for the ingenuity of the ideas and implications of their success, we also hear the remark "But how are you going to program such a beast?", thereby implying that the software problem remains once the hardware is designed. Similarly, several attempts are being made at parallel programming language design; one hears the remark "What we really need are ways of thinking and problem solving that incorporate parallelism". This reaction stems from the view that a programming language is merely a notation and new developments in programming methodology are essential for the proper use of the next generation of computers.

We have embarked on a project to explore in tandem programming methodology and programming language design, allowing each to condition and influence the other. Thus, we start with a range of what we consider interesting problems, explore how we would like to express a range of solutions to these, paying attention to the methodology of algorithm development. In the course of doing this we evolve both our language design and our programming methodology. This is in contrast to the more usual approach of making the language reflect the architecture of a given machine and abstracting away selectively from it. Thus our language is rife with functional primitives rather than with machine-oriented primitives. Our aim is to make the tasks of programming and tuning for performance be cognitively simple and error-proof.

Most current parallel programming languages are designed with a sequential programming language as the base language and then adding constructs that allow parallel execution. It is therefore not surprising to see researchers attempting to analyze serial algorithms to discover (or uncover) hidden parallelism. Wisdom has it, and experience proves it, that there is only limited concurrency that can be exposed by examining extant algorithms. One needs to start afresh with the problem and develop parallel algorithms in order to witness greater parallelism.

We are experimenting with a programming language that has an *implicitly* parallel applicative language as the base language, and then we introduce constructs that *inhibit* parallelism. The base language uses pure LISP [25] as a foundation and blends in interesting features of Prolog [23] and FP [5]. *Proper utilization of available machine resources is a crucial concern of programmers.* This is an outstanding problem for both functional programming and logic programming. We introduce several techniques of controlling the behavior of functional programs without changing their meaning or functionality: (i) program annotation with constructs that have benign side-effects; (ii) program transformation; and (iii) adaptive scheduling. This combination yields us a *semi-applicative* programming language and an interesting programming methodology.

Section 2 describes the base language SALT and the annotation language PEPPER. The final section 3 summarizes our efforts in the area of context-free parsing and discusses the derivation of several recognition algorithms, all related to the recognition method of Cocke-Kasami-Younger [20] and that of Earley [14]. *All of these algorithms reveal more concurrency than was apparent at first glance.*

1.1. Motivation

We aim at developing a programming language and programming methodology that allow effective use of medium-scale, medium-grain parallelism; support correct program development, and allow effective, error-free control of program behavior through a variety of means.

As an initial set of problems to study in the project, we are investigating search algorithms (alpha-beta, branch and bound, backtrack), constraint propagation and marker propagation algorithms, constraint satisfaction and relaxation algorithms and parsing algorithms. This paper, however, is concerned solely with recognition algorithms for context-free grammars in Chomsky-Normal-Form.

As an illustration of our programming methodology, we start with the specification of a recognizer for context-free grammars in Chomsky-Normal-Form (CNF) and derive by transformations a variety of different purely applicative parallel recognition algorithms. We then introduce program annotations and display semi-applicative algorithms. In one algorithm we indicate how adaptive scheduling can control the behavior exhibited by the algorithm. Thus we hope the reader observes the use of transformations, annotations and scheduling as means of controlling program behavior.

1.2. Experience

The current Butterfly Lisp project [2] is building a parallel Common Lisp for the Butterfly multiprocessor [8] starting with Scheme [1], in cooperation with MIT researchers. This effort represents an important near-term approach to providing parallel computation for AI research. The Butterfly Lisp project at BBN has provided us with an opportunity to gather experience in developing parallel algorithms, coding programs and testing them. We have programmed several algorithms in applicative subsets of both MultiLisp [17] and Scheme [1] and have tested them on the Butterfly multiprocessor and also with simulated parallelism on the VAX-11/780. The range of algorithms with which we have experimented includes:

- Combinatorial search algorithms

  - N-Queens problem; incorporating a variety of modes of parallel computation, *e.g.* parallel vs serial evaluation of partial board placements; parallel vs serial forking of search tree nodes.

  - Dynamic programming for an optimization problem; employing different ways of setting up initial tasks, different dependencies among tasks, different ways of utilizing data flow and caching operations.

- Recursive non-search algorithms

  - Parallelism as well as pipelining for elementary recursive functions such as Fibonacci, Ackerman and QuickSort.

This experience has highlighted several key issues that need to be addressed in the development of parallel algorithms. *Generating an appropriate level of concurrency in an algorithm is difficult.* It is possible to generate too little concurrency or too much concurrency in a recursive symbolic program, especially in search algorithms. Moreover, current languages make the programmer indicate explicitly when to spawn new tasks and when to carry out computations serially. The decision to spawn tasks should depend to a great extent on availability of processors, memory availability and other run-time characteristics. Such decisions cannot be made at program composition time, or even at compile time. *Scheduling decisions clearly exceed the capability of the programmer.*

The program development and testing environment should enhance the ability of the programmer to achieve the goal of proper utilization of machine resources. The environment that we envision lays less emphasis on program debugging and encourages correct program development and effective control of program behavior through a variety of methods all of which preserve correctness of the program. Furthermore, it gives the programmer *a simpler computational model* than models prevalent today.

2. Detailed Discussion of our Approach

Most efforts at developing a parallel language start with a sequential language (Lisp, Pascal, Algol) and introduce concepts relevant to parallel execution (Fork/Join, Task, Task groups, Critical region, Monitors, Semaphores, Interprocess communication, event-wait/signal etc.). [See [4] for a survey.] These allow the programmer to state and control parallelism explicitly. In general, they create a very complex model of computation, making the task of the programmer difficult and error prone. Approaches similar to that of Concurrent Prolog [31] present a simpler computational model, but restrict the expressive power of the language by constraining the computational model too much. They are also limited to using only first-order predicates and have unclear techniques for using evaluated functions and for introducing user-defined control structures. Approaches like that of Qlisp [15], MultiLisp [17] and parallel CommonLisp [2] allow sequential code interspersed with parallel constructs. The extent of *residual sequential code* in the program limits the speed-up possible in programs [22], following Amdahl's law[1].

Our approach is based on the use of a side-effect free programming language, which has *implicit* concurrency everywhere. Our annotation language allows the programmer to control concurrency explicitly by adding annotations to the program text. The annotations form a small set, each member of which guarantees that the meaning (clarity and semantics) are unchanged by its introduction. Each annotation can affect the *behavior* of the program, i.e. alter its runtime, space utilization, reuse of computed results, total work, extent of concurrency and extent of inter-task communication. *We believe that this approach combining implicit parallelism and explicit control will reduce the risk of introducing bugs in the process of tuning programs for performance*, as well as providing an antidote to Amdahl's law.

[1] If S is the fraction of residual sequential code, and N the number of processors available, the maximum speed-up is $N \cdot (1+(N-1) \cdot S)^{-1}$ which is bounded by 1/S when N is very large. For example with N=1000 and S=0.1, 0.01 and 0.001 we get the maximum speed-up to be 9.91, 90.99 and 500.25 respectively. Squeezing out the residual sequentiality might, unfortunately, end up being the analog of trying to get the last bug out, i.e. an unending enterprise each succeeding step consuming disproportionately larger effort on the part of the programmer. Clearly, an alternative approach is called for!

2.1. Computation Model

The computational model that we advocate to the programmer might be termed unlimited virtual parallelism. The program runs as a large set of concurrent tasks, which may be run in any arbitrary order, subject to certain constraints. Central to the system is a scheduler that maps the unlimited virtual parallelism to the limited physical resources of the machine. Virtual parallelism is analogous to virtual memory systems that allow the programmer a vast amount of virtual space, and are aided by a heuristic memory management subsystem. The ordering of task execution is constrainted by three graphs mentioned below. The programmer views control of the computation in terms of the same three] graphs over the set of tasks: the Spawning graph that specifies which tasks are spawned by which other task; the Communication graph that specifies which tasks produce values consumed by which tasks; and the Precedence graph that specifies which tasks can be begun after the termination of which other tasks. The set of program annotations allow flexible control over the behavior of the program, and are conceived in terms of changes to these three graphs.

There are three styles of computation [34] that are quite well understood presently. Control Flow, is viewed in terms of a control token that passes around the program. Only the segment of the program with the control token is in control and thus can be active. For parallel execution, control tokens are copied. When a task terminates, its copy of the control token disappears. In Data flow, all segments of the program are active, pending only the availability of the data they are to consume. In Demand Flow, each (statement or expression) propagates, in parallel, a demand token that activates other statements that can compute the demanded results. In our language we unify these styles of computation allowing the programmer to not only choose a preferred style but also to mix them as s/he sees fit.

2.2. Base language: SALT

SALT is an applicative functional language free of side-effect causing constructions. An initial design document [33] for SALT is available, and it demonstrates the ease with which an interesting range of algorithms may be expressed in SALT. SALT is based on the pure Lambda calculus and adopts a functional syntax. It includes the usual facilities for function definition (using conditionals and recursion), naming, both object and function variables, anonymous functions in the form of Lambda abstractions, and higher-order functionals (like Maps, which provide data-parallelism).

Like Scheme, it adopts lexical scoping. From FP, it gets a suitable set of program-forming operators, like the functional Insert, the composition operator (pipelining parallelism) and structured function lists (work-splitting parallelism). We have argued elsewhere [33] that functional programs with parallelism of the kind mentioned above make it simple to arrive at only limited forms of parallelism. Namely, parallel activity where the spawning graph, communication graph and precedence graph all mirror the structure of the function call graph are easy to describe. One has to resort to very complex constructions to describe arbitrary communication between subtasks or arbitrary precedence structures. We have found that introducing the idea of the logical variable into the language and allowing these to be shared is a very useful extension which allows us to achieve arbitrary types of parallelism more easily. To this end, from PROLOG we borrow and adapt three important ideas:

1. Functional call semantics is not in terms of binding the value of actual argument to formal parameters, but in terms of unification of the *value of* actual arguments with *formal parameters*, thus affording *two way communication* between the caller and the called function;

2. Constructors are used to define compound parameters; they facilitate assembling and disassembling structured arguments and results,

3. Multiple definitions for a single function are allowed; they differ in their parameter structures, allowing the compiler greater freedom in testing/selecting function bodies to execute.

Shared logical variables among tasks are permitted in SALT, their values can be refined in stages by successive unifications. The communication graph is determined by the shared variables as well as by function-call-return graph. Thus shared logical variables facilitates multi-way communication among tasks without limiting communication to parent/child pairs. These ideas are elaborated in the companion technical report [33]. We do NOT use the backtracking control structure of Prolog. Failure of unification is considered an error condition[2].

For the limited purposes of this paper, that is, to illustrate aspects of semi-applicative programming and to show the derivation of several context-free recognition algorithms, we shall not need many of the properties of SALT. The algorithms in section 3 are exhibited in an informal notation, using fairly standard programming concepts. Some of the notation will be explained as they are introduced.

[2] We have not explored the issue of error-handling yet.

The reader should simply bear in mind that in all cases where "For" loops, "Subset" and "Union" expressions are introduced, they are not serial; all these constructions are parallel forms. A serial "For" will be introduced when needed and written *"(For <var> in <list>* *serially* *...)"*. In the examples in this paper we do not use the unification semantics in any essential way; but we find the use of structured parameter lists quite convenient for disassembling and assembling structures, and we find they enhance readability and conciseness.

2.3. Resource control in applicative programs

Because we use an applicative language which specifies computations without side-effects, we gain the advantages of simple clear semantics and generous opportunities for concurrency. Of course, what we lose is direct control over the computational behavior of the program. *Proper use of parallel resources is a crucial concern of programmers*. This is an outstanding problem for functional programs and logic programs. We introduce several methods of altering the *behavior* of programs without altering their correctness or the functionality of what they compute. Among the techniques we are exploring are

2.3.1. Program annotations

Program annotations [11] add resource control features to the source-level algorithm. We have developed an initial design for an annotation language called PEPPER. Annotations in PEPPER include precedence control, function call/result caching [10, 21], and lazy (or demand-driven) evaluation. Our initial design document describes PEPPER annotations and gives several examples of their use. A key feature of PEPPER annotations is that their introduction will not alter the functional value of a program, and will affect only its run-time behavior.

2.3.2. Program transformation

Program transformation [6, 9, 26] of the source-level algorithms written in SALT is essential. There are two quite different reasons for considering program transformation. Firstly, given that PEPPER annotations are to be added to the text of a program, it is evident that *textually different but functionally equivalent programs offer different opportunities for adding annotations*; and hence provide different opportunities for achieving different behaviors. A programmer can make full use of annotations only in conjunction with the ability to do program transformations. [An example of this is presented in [33]; see Synchronizing Multiple Streams]. Secondly, *different programs yield different data-dependency orderings*, thus allowing differing amounts of concurrency. Thus, program transformation is a technique for controlling the available concurrency in a program.

2.3.3. Adaptive scheduling

Adaptive scheduling can yield improved behavior with repeated runs of the same program. Adaptation requires monitoring and measuring run-time characteristics of SALT+PEPPER programs in order to make scheduling decisions dynamically. In working with virtual parallelism, the program only expresses "available" concurrency without mandating what in fact will execute concurrently with what. The scheduler converts the available concurrency into physical concurrency, making its choices for running tasks by considering available resources, resource requirements and other attributes of tasks. The scheduler typically is a heuristic procedure and does not yield optimal behavior in all cases. Thus, another avenue open to the programmer is to tune some parameters of the scheduler to alter the behavior of the program. We feel this method of control is "indirect", and while that is necessary, it is not likely to be sufficient. It may be useful to construct an adaptive scheduler that tunes its parameters based on empirical measurements.

Consider an example, where a search algorithm has been programmed in terms of a set of tasks that spawn other tasks to span the search space, and a corresponding set of testing and evaluation tasks that examine proposed solutions, but spawn no new tasks. Empirically it may be feasible to identify which tasks terminate without spawning new tasks, and which tasks spawn new tasks but wait for them to finish before they themselves can finish up. A reasonable scheduling strategy is to always give precedence to the evaluation tasks.

2.4. Annotation language: PEPPER

All programs written in SALT permit concurrent evaluation of all subexpressions, generating the possibility of fine-grained parallelism. Pepper annotations are introduced around expressions; they restrict the computational behavior of the expression, but compute correctly the value of the expression. The name PEPPER is chosen to symbolize the idea that functionally correct SALT programs will be sprinkled with PEPPER annotations to derive the desired computational behavior.

The main types of annotations include precedence control and caching. Precedence control annotations introduce conditions that must be met for the triggering of a task, and thus alter the precedence graph for the program. There are three types of precedence control constructs. Control sequencing restricts one task to wait for the completion of one or more other tasks. Data flow sequencing makes function applications wait for the delivery of (all or some of) their arguments. Suspension annotation causes the evaluation of forms to be postponed till their values are demanded; such suspensions provide one form of lazy evaluation of arguments.

Caching permits expression values to be cached and reused to avoid redundant computation. In several examples we will show an exponential time algorithm reducing to polynomial time, resulting from the proper use of caching. A number of recursive functions can be transformed from the usual top-down version to perform bottom-up computations [10] by means of table build-up. In some cases, the top-down version may cycle endlessly whereas the bottom-up version may efficiently compute and terminate.

We describe here only three caching annotations that will be used in this paper. The remaining annotations are described and illustrated in the companion technical report. Function calls, not merely results, need to be cached in a concurrent environment, to catch all redundant calls. Otherwise, after the first invocation, if a second call is made before the first one completes, a duplicate invocation will be made. *(Cache (f . args))* replaces a normal function call with a cached call. If there is a cache entry and if it contains a result, the result is retrieved and returned as the value of this form. If there is no cache entry, a cache entry is created with no result and the function f is invoked with its arguments. When the function completes the result is posted in the cache entry and the result is also returned to the invoking form. The invoking form waits for the delivery of results. If there is a cache entry which contains no result, this is an indication that computation is in progress concurrently. In this case, the invoking form waits for delivery without calling the function f. This discipline for caching is appropriate for concurrent computation and differs subtly from the standard memo-izing operation which caches only results after they are computed, not the function calls. Another caching construct we use is of the form *(Lookup (f . args))*. This is a straight table lookup without any need to perform checks or waits. The table build-up is either done ahead of time, or there is enough serialization in the algorithm to guarantee that the entry will be ready when the lookup is done. Another caching annotation is mentioned here but is developed more fully in the final example, the Earley parser. *(CacheDefault (f . args) DefaultValue)* is a loop-breaking construct. It behaves like Cache, except that under specified conditions (when a loop is detected), it will not wait for delivery of values, but will return the DefaultValue.

3. Research Results: Derivation of recognition algorithms for CNF grammars

In this section we demonstrate the derivation of several recognition algorithms for context-free grammars. The initial specification is a standard mathematical definition for what a recognition algorithm should do. In several steps of transformation, we derive a variety of parallel recognizers (see Figure 3-1). In the literature there are two well-known parsing algorithms. Cocke, Younger and Kasami, independently reported on the development of a bottom-up parser, called CKY, that works on a restricted form of grammars called the Chomsky-Normal-Form. Earley published an algorithm that works with unrestricted context-free grammars and parses an input string in a top-down fashion using left-context to limit search. These algorithms are described in this section. Most of us, including the author, have admired the cleverness of such algorithms. One of the things we do in this section is to demonstrate that these algorithms have real close affinities to each other and that they can be arrived at systematically. Similar derivations of various SORT algorithms have been previously presented in the literature. All of these have been successful only at deriving a *few of the already known sort algorithms; none of these derive any* new SORT algorithm. Can one readily adopt these transformational techniques to derive new algorithms? Our attempt hopefully convinces the reader that new parallel algorithms can be derived systematically.

The reader will notice that the transformations used in the derivation are only informally stated, even though the program being derived is precisely written down at each step. Furthermore, the transformations are fairly large, and would take substantial analysis and rewriting capabilities to support in a semi-automated system. We do not hint or suggest at this time that such a derivation sequence can be automated. By using these examples and other examples to be developed in the future, we hope to arrive at a conception of what a programming environment would be that could assist us in the development of semi-applicative parallel algorithms.

3.1. Preliminaries: Grammar, Normal Form and Derivation

We start with the standard definition of Chomsky-Normal-Form grammar, Derivation and Recognizer [18].

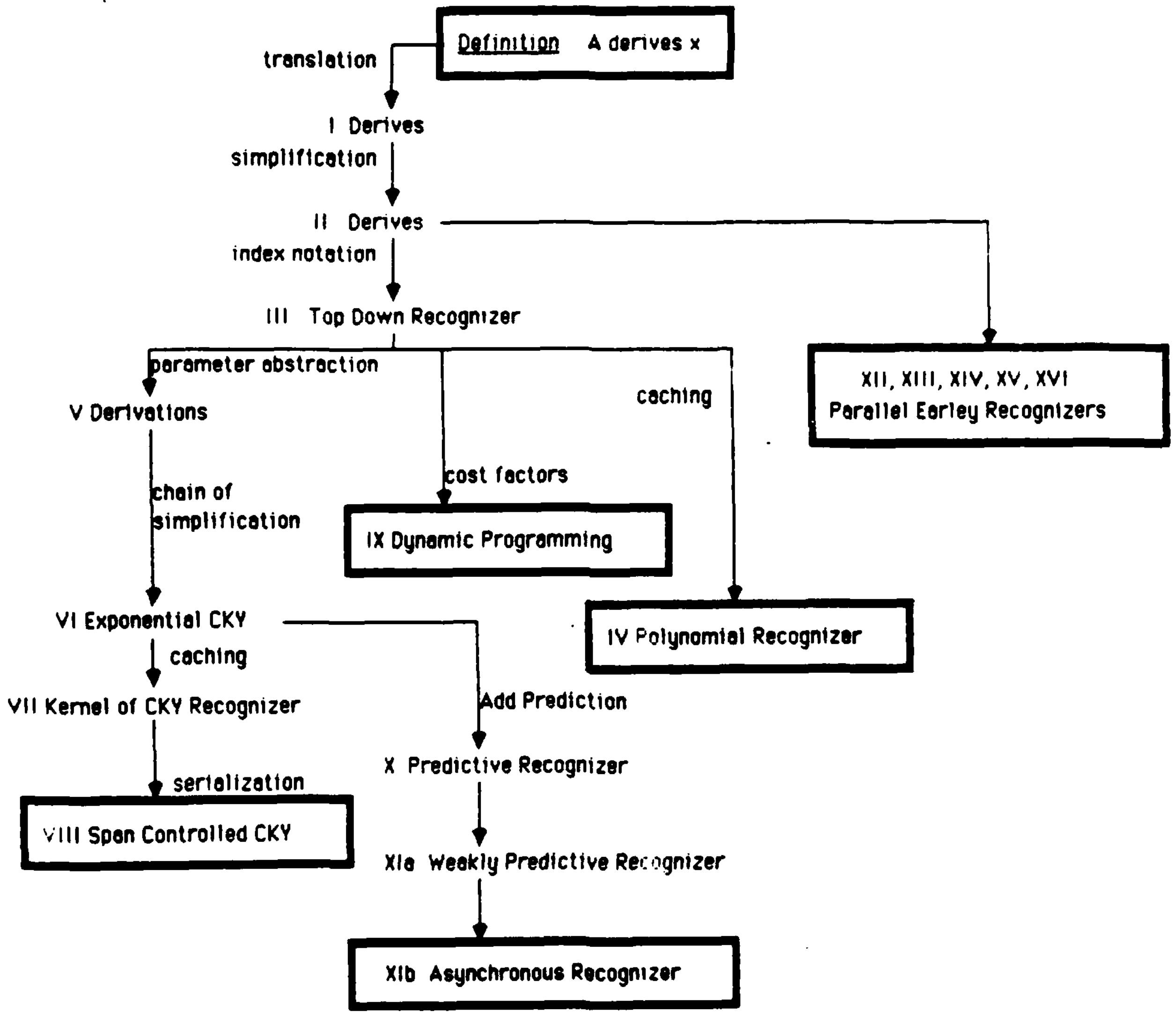


Figure 3-1: Chart showing transformational derivation of different algorithms for context-free recognition using Chomsky Normal Form grammars

A *Chomsky Normal Form grammar*, abbreviated CNF, henceforth simply grammar, is identified in terms of

T: a set of terminal symbols,
N: a set of nonterminal symbols,
V: the set of terminal and nonterminal symbols, T+N,
S: a distinguished nonterminal symbol,
P: a set of productions, comprised of two sets UP and BP.

Each production in UP, a unit production, is of the form

$A \rightarrow x$ where $x \in T$, $A \in N$

Each production in BP, a binary production, is of the form

$A \rightarrow BC$ where $A,B,C \in N$

We use the notation
V*: for the set of strings over V,
T*: for the set of strings over T.

A unit production has a single terminal symbol for its right-side and a binary production has a pair of nonterminals. A one step derivation involves starting with a string uAv where $u,v \in V^*$ and $A \in N$, and using a unit production $A \rightarrow x$, deriving the string uxv (notation: uAv=>uxv); or using a binary production $A \rightarrow BC$, deriving the string uBCv (notation uAv=>uBCv). A Derivation is a chain of one step derivations (notation starting-string=>*ending-string). The language generated by the grammar is the set of terminal strings that can be derived starting with the distinguished symbol S.

Any Context-free (CF) grammar can be converted to an equivalent grammar in CNF. From a practical point, however, when a CF grammar is converted to CNF, its size (number of productions) may grow exponentially. Thus, practical algorithms need to work directly with CF grammars. The development of our algorithms is simplified by the use of CNF. However, there is nothing essential in the character of CNF that we depend upon; and thus we believe all the algorithms may be generalized suitably to work with unrestricted context-free grammars. Furthermore, we shall be writing algorithms for "recognizers" which only decide whether the input string is in the language generated by the grammar; they do not return a parse structure. The published algorithms also adopt this route; and they all use a fairly simple technique for reconstructing the parse tree. We shall not elaborate on this any further and confine our attention to the recognition problem, that is, the computation of the predicate "A derives x".

A <u>recognizer</u> is a function (Recognize x) that takes the input string x and returns a boolean value which is the correct answer to the question "Does S derive x?" We start with a more general function, "Does A derive x" where S, the distinguished start symbol, has been replaced with an arbitrary nonterminal, A. Thus, (Recognize x) = (Derives 'S x).

3.2. Derivation of a parallel top-down recognizer

For any nonterminal $A \in N$ and terminal string $x \in T^*$ we define

<u>A derives</u> x

```
    if either there is a p ∈ UP such that p = A -> x
          or there is a p ∈ BP such that p = A -> BC,
                 and a partition x=yz, y and z nonempty
                 and B derives y and C derives z.
```

Since, we have started with a recursive definition of the predicate Derives, it is quite straightforward to turn the specification directly into an effective algorithm (it will always terminate and provide the correct result).

<u>Algorithm I</u> *Top-down recognizer*

```
    Invoke (Derives 'S input)
    Define (Derives A x) =
           (or (member [A -> x] (UP A))
               (For Some [A -> BC] in (BP A)
                    (For Some [y,z] in (split x)
                         (and (Derives B y) (Derives C z)) )))

    with (split x) =
          case |x| < 2  {}
          case |x| = 2  {[(first x), (second x)]}
          case |x| > 2
               (Union {[(first x), (rest x)]}
                   (For [y,z] in (split (rest x))
                        collect [(first x)||y , z]))
```

Notation: We are using square brackets to denote data structuring; curly brackets for sets; || for string concatenation; first and rest functions over strings; |x| denotes length of string x; (UP) for the set of unit productions; (UP A) for the set of unit productions with A as their left-hand-side; (BP) without any arguments to denote all binary productions; (BP A) for all binary productions with A as their left-hand-side. We may introduce a structure, e.g. [y,z] above, where one would normally use a variable. Unification of such a structure produces structure disassembly. Notice that the "For" over the set is a parallel construct. We prefer to use the "For" construct over the "Map" construct for expository convenience.

The above algorithm is guaranteed to terminate since each recursive call to Derives involves strings that are strictly shorter. Thus the depth of recursion is no greater than the length of the input string.

The non-determinism involved in the use of "Or" is eliminated by introducing a deterministic case analysis. Notice also that the nesting of the two search constructs "For Some" is unnecessary since neither generator depends upon the other. Thus we rewrite Algorithm I to expose greater concurrency. The construct *"(For Some [<var1>,<var2>] in <set1> cross <set2> ...)"* is used to write unnested loops with var1 taking values in set1 and var2 taking values in set2.

Algorithm II *Exponential top-down recognizer*

```
Define (Derives A x) =
       case |x| = 1 (member [A -> x] (UP A))
       case |x| > 1 (For Some [[A -> BC],[y,z]] in (BP A) cross (split x)
                                (and (Derives B y) (Derives C z) ))
```

Though algorithm II is effective, it can take time exponential in the length of the input. In fact, the following recurrence characterizes the time to recognize a string of length n with p productions: $T(n) = \text{constant} + p * \text{sigma}(1 \leq k < n)[T(k)+T(n-k)]$.

Recognizing that much of the effort in this algorithm is wasted in redundant calls to Derives with the same arguments, we introduce caching. Prior to doing that, let us transform the string descriptions (and the Split function) to take advantage of array capabilities of current machines. All of the string arguments given to Derives are substrings of the original input string. Thus substrings can be referenced by a pair of indices marking the beginning and end. Indices range over (0,n) each index value referring to the space after the ith character and before the (i+1)st character. Thus the entire input string is the pair [0,n]. The unit string involving the ith character is [i,i+1]. Thus all non-empty substrings are referred to by [i,j] with i<j.

Algorithm III *Top-down recognizer using indices*

```
Invoke (Derives 'S [0,n])
Define (Derives A [i,j]) =
       case j-i = 1 (member [A -> (input i j)] (UP A))
       case j-i > 1 (For Some [[A -> BC],[[i,k],[k,j]]] in (BP A) cross (split [i,j])
                         (and (Derives B [i,k]) (Derives C [k,j])))
with (Split [i,j]) =
       (For k in (i+1 to j-1) collect [[i,k],[k,j]])
```

Notice that "Split" has been converted into an iterative algorithm and string operations have been replaced with index manipulation. The form *(input i j)* refers to the substring of the input string. We now introduce caching of results for "Derives". Since the algorithm has concurrency, caching should be done both for *invocations* of Derives as well as for its *results*. By caching invocations and not merely results, in a concurrent setting we avoid duplicate invocations, as explained before.

Algorithm IV *Polynomial Top-down recognizer with Caching*

```
Define (Derives A [i,j]) =
        case j-i = 1 (Member [A -> (input i j)] (UP A))
        case j-i > 1 (For Some [[A -> BC],[[i,k][k,j]]] in (BP A) cross (split [i,j])
                           (and (Cache (Derives B [i,k])) (Cache (Derives C [k,j]))))
```

This brings to conclusion one line of development, and gives us a top-down recognizer that uses caching to avoid redundant computations. The entire algorithm is written with no explicit serialization. Various calls to Derives are invoked without explicit synchronization, and if they are looked up before the results are ready, the invoking form will wait for delivery of results. The wait lists so constructed form a data-flow dependency graph and thus provide an implicit form of data-flow scheduling. The algorithm, with data-flow scheduling, exposes more parallelism than the the span-controlled recognizer which is usually presented as the CKY algorithm. This span-controlled algorithm is derived further below, see Algorithm VIII. The maximum time taken by this algorithm IV is related the maximum number of distinct calls to Derives. This number can be seen to be the product of the number of distinct first argument values, which is p, the number of productions, and the number of distinct second argument values, which is the number of substrings of the input string of length n, that is, $n*(n+1)/2$. Each call takes $p*n$ time at most. Thus the worst-case serial timing of the algorithm has been reduced from exponential to p^2n^3. Given unbounded computational resources, this algorithm will run in time linear in n, which is the depth of the data-flow graph. This provides an upper and lower bound for the actual performance for this algorithm on a multiprocessor.

3.3. Derivation of a parallel bottom-up recognizer

We continue a different line of development ending with a parallel version of the Cocke-Kasami-Younger algorithm. This latter algorithm computes bottom-up the set of nonterminals that can derive each substring of the input. The CKY algorithm is usually presented in the following sequential form:

<u>Algorithm Serial CKY</u>

```
Declare Procedure CKY(R);
Declare R[0:n,0:n] set of nonterminals;    *Recognition matrix*

  *Handle unit characters*
for j = 1 to n do
  R[j,j+1] = Set of nonterminals A such at A → (input j j+1) ∈ UP;
  *Handle binary composition*
for l = 2 to n do      *l is the span length of substring*
  for i = 0 to n-l do  *i is the start position of substring*
    begin
      j = i+l;  *j is the end position of substring*
      R[i,j] := empty-set;
      for k = i+1 to j-1 do  *consider all splits into non-empty parts*
        for B in R[i,k] do   *consider all interpretations of [i,k]*
          for C in R[k,j] do *consider all interpretations of [k,j]*
            begin     *combine BC to get interpretation A for [i,j]*
              R[i,j] := R[i,j] union Set of nonterminals A such that
                                            A → BC ∈ BP.
            end;
    end;
end Procedure CKY;

Declare R[0:n,0:n] initial empty-set;
Call CKY(R);
Test 'S ∈ R[0,n];
```

When such an algorithm is presented, it needs to be explained and also proved. The first loop handles unit characters of the input string and applies all unit productions; entering their interpretations in the corresponding cells R[j,j+1]. The outermost loop (variable l) in the second section of the program is systematically increasing the span of the substring considered from 2 to n. The loop variable i ranges over all possible starting positions for a substring of length l; j is then set to the end position of such a substring. To arrive at interpretations for the substring [i,j] all possible splits are considered (loop variable k) and the results combined. The algorithm works bottom-up arriving at interpretations for substrings of greater length. Since the interpretations for substring of length l depends only on substrings that are shorter, we know that the needed values will be already be computed. Despite proper explanation of how it works and the proof, one is left wondering what the inventive step in the creation of the algorithm was and how the details were structured to get it correct. For example, are the loop indices bounded correctly? Is the initialization needed? What steps can be done in parallel?

We fall back to Algorithm III which is still in functional form and propose a series of transformation steps that culminates in a parallel version of the CKY algorithm.

Algorithm III *Top-down recognizer using indices*

```
Invoke (Derives 'S [0,n])
Define (Derives A [i,j]) =
        case j-i = 1 (member [A -> (input i j)] (UP A))
        case j-i > 1 (For Some [[A -> BC],[[i,k],[k,j]]] in (BP A) cross (split [i,j])
                          (and (Derives B [i,k]) (Derives C [k,j])))
with (Split [i,j]) =
        (For k in (i+1 to j-1) collect [[i,k],[k,j]])
```

The first thing to change is that instead of querying (Derives A [i,j]) we would be querying A ϵ (Derivations [i,j]). In the body of definition of Derives, we have occurrences of the variable A, whose domain is the set N of nonterminals. Since the variable A is not going to be a parameter to "Derivations", we surround the expressions for both cases in Algorithm III by "(subset N (Lambda (A)...))" where N is the set of nonterminals. "Subset" takes a set and a predicate, returning the subset with elements for which the predicate is true.

(Derives A [i,j]) = A ϵ (Derivations [i,j])

Algorithm V

```
Define (Derivations [i,j]) =
        case j-i = 1
             (subset N (Lambda (A) (member [A -> (input i j)] (UP A))))
        case j-i > 1
             (subset N
                 (Lambda (A)
                     (For Some [[A -> BC],[[i,k],[k,j]]] in (BP A) cross (split [i,j])
                         (and (member B (Derivations [i,k]))
                              (member C (Derivations [k,j])))) ))
```

The main point of the algorithm is generating productions paired with splits of the string and testing to see if the production is applicable. We shall focus our attention on the expression involving the second case, j-i>1, since there is nothing interesting in the transformation for the first case, j-i=1; it merely follows a similar but simpler line of development. We separate the loop over the cross-product set, and create two for-loops, in preparation to eliminate the loop over the productions.

```
(subset N
    (Lambda (A)
        (For Some [A -> BC] in (BP A)
            (For Some [[i,k],[k,j]] in (split [i,j])
                (and (member B (Derivations [i,k]))
                     (member C (Derivations [k,j]))) ))))
```

Now we use our knowledge of the fact that the outermost loop "subset" ranges over all nonterminals A in N, and that the inner loop ranges over all productions with a given nonterminal A. We combine and collapse the two loops. To do this we eliminate the search through N via the expression (subset N ...), and replace (BP A) with (BP). We get:

```
(Union for [[i,k],[k,j]] in (split [i,j])
    (For [A -> BC] in (BP)
        collect A if (and (member B (Derivations [i,k]))
                          (member C (Derivations [k,j]))) ))
```

The above algorithm uses the two membership predicates as tests and the set of productions as the generator. Now we interchange the testing and generating loops in order to isolate a fragment of the algorithm that is dependent only on the grammar and not on the input string. We carrying this out by turning the two membership tests into generators and burying the loop over productions in a test.

```
(Union for [[i,k],[k,j]] in (split [i,j])
    (Union for [B,C] in (Derivations [i,k]) cross (Derivations [k,j])
        (For [A -> BC] in (BP) collect A))
```

The set of nonterminals A which derive BC written as "(For [A —> BC] in (BP) collect A)" is now abstracted as a function "(Binaries B C)".

```
(Union for [[i,k],[k,j]] in (split [i,j])
    (Union for [B,C] in (Derivations [i,k]) cross (Derivations [k,j])
        (Binaries B C)))
```

Let us unfold the definition of (split [i,j]) and simplify it to get:

```
(Union for k in (i+1 to j-1)
    (Union for [B,C] in (Derivations [i,k]) cross (Derivations [k,j])
        (Binaries B C)))
```

This yields the last step in the development of the CKY algorithm. We have focused our attention on the case where j−i>1. A similar development is done for the case where j−i=1, producing a function "(Units x)" which returns a set of nonterminals which have the production [A —> x] in UP.

<u>Algorithm VI</u>

```
Invoke (member 'S (Derivations [0,n]))
Define (Derivations [i,j]) =
       case j-i = 1 (Units (input i j))
       case j-i > 1 (Union for k in (i+1 to j-1)
                       (Union for [B,C] in (Derivations [i,k]) cross (Derivations [k,j])
                          (Binaries B C)))
  with (Binaries B C) = (For [A -> BC] in (BP) collect A)
       (Units x) = (For [A -> x] in (UP) collect A)
```

We still have to introduce caching. Since the functions "Binaries" and "Units" depend only on the grammar and not on the input string itself, they can be precomputed and set up in a table. They can be queried via a table lookup using (Lookup ...); no testing or waiting is required. We cache "Derivations" to provide us the recognition table. This yields a pure bottom-up recognizer, presented below, with its name changed from Derivations to CKY.

Algorithm VII *The kernel of Cocke-Kasami-Younger algorithm*

```
Invoke (member 'S (CKY [0,n]))
Define (CKY [i,j]) =
        case j-i = 1 (Lookup (Units (input i j)))
        case j-i > 1
            (Union for k in (i+1 to j-1)
               (Union for [B,C] in (Cache (CKY [i,k])) cross (Cache (CKY [k,j]))
                  (Lookup (Binaries B C)) ))
```

When we invoke the recognition test (member 'S (CKY [0,n])) it will successfully invoke, in several stages of parallel demand flow, all the entries of the recognition table. Whenever a table entry is not ready the computation will wait for the delivery of values. Thus the scheduling of parallel activation of all table entries is limited by the demand flow and data flow ordering alone. We get greater concurrency than when the standard control flow version of the CKY algorithm is used.

The standard serial control flow version of the serial CKY algorithm involves no explicit calls for synchronization; the parallel version shown above requires the overhead of checking and waiting. We can proceed to eliminate this by serializing the computation to guarantee that values will be available when looked up. Since each table entry for string [i,j] depends only upon values of table entries of its substrings, ordering the computation by increasing string length is possible. We trade off some of the concurrency available in the data flow version, and we introduce control flow. We eliminate the demand flow and data flow dependency and replace the calls to Cache by calls to Lookup.

Algorithm VIII *Span Control Ordering*

```
Invoke (CKY-span-control n) and (member 'S (CKY [0,n]))

Define (CKY-span-control n) =
       (For span in (1 to n) serially
            (For i in (0 to n - span)
              do  (CKY [i,i+span])))
with (CKY [i,j]) =
     case j-i = 1 (Lookup (Units (input i j)))
     case j-i > 1
         (Union for k in (i+1 to j-1)
            (Union for [B,C] in (Lookup (CKY [i,k])) cross (Lookup (CKY [k,j]))
               (Lookup (Binaries B C)) ))
```

We have added a serial control flow that systematically increases the span of substrings considered, starting at 1 and ending with n. One can imagine this (as is usually explained in text books) as starting with the main diagonal of the CKY recognition table and filling the next upper diagonal after the one below it is filled in. Notice that the inside For loop in CKY-span-control is a parallel loop. All the n-span+1 calls CKY spawned for each value of span are allowed to execute in parallel and must terminate before the next value of span is executed. This control flow ordering guarantees that whenever CKY does an invocation of the CKY calls contained in its body, the corresponding results it is going to lookup will have been computed and made available. Therefore, we have replaced "Cache" calls with "Lookup" calls which do not require any synchronization waits. All "table build-up" methods cited in [10] rely upon such an ordering to avoid synchronization waits.

The innermost "Union" can be computed in time independent of the string length; the effort it requires is a function of the number of nonterminals, p. The components of the outer "Union" can all be invoked in parallel, as can all the invocations of i for any given span. The time taken by this algorithm, given no bounds on the number of processors, only grows linearly with n corresponding to the serial ordering of the control flow.

3.4. Derivation of a dynamic programming technique

By introducing a fixed cost factor with each production in the grammar, and defining the cost of a derivation to be the sum (or maximum) of the productions used in it, we now turn the recognition problem into an optimization problem. In this subsection we display a dynamic programming algorithm that finds the least cost for a derivation (omitting the detailed development of this algorithm). Clearly, if there is any derivation of the input string from S, there must be a least cost one. We merely avoid seeking all possible recognitions.

When a unit production is used, the algorithm should return the cost associated with that production. When a binary production is used, the algorithm should compute the cost of the derivation by adding in the two minimum costs of the sub-derivations to the cost of the binary production. We presume modified functions BP and UP which return a set of pairs, each pair containing a production and also its cost. We also rename Derives to MinCost, to reflect its new meaning.

Algorithm IX *Dynamic Programming for minimum cost parse*

```
Invoke (MinCost 'S input)
Define (MinCost A x) =   ; returns a cost
       case |x| = 1 (Cache (UnitCost x))
       case |x| > 1 (For [[[A -> BC],ABCCost],[y,z]] in (BP A) cross (split [i,j])
                          min (+ ABCCost (Cache (MinCost B y)) (Cache (MinCost C z))))
       with (UnitCost x) =
            if (member [[A -> x],AxCost]] (UP A))
            then AxCost else +Infinity
```

To retrieve the cost of a production, we are altering the test for membership to include a cost variable. When the test for membership is invoked, the cost variable is uninstantiated and has no value; a successful result not only tests to see if the production is there, but also unifies the cost variable with the stored cost of the production. Thereby, the cost is retrieved. Here is one case where we have stepped outside of the pure functional programming style and have used the capability of SALT to do unifications. Without such a facility this algorithm can still be expressed in a pure applicative style, but it would not be as perspicuous, nor will the relationship between algorithms II and IX stand out clearly. Note also the fact in converting Derives which returns a boolean value, to MinCost which returns a cost, we also introduced the mapping that false value maps to the cost of +Infinity. This algorithm runs in time proportional to n^3 in the serial worst-case. It runs in time proportional to n*(log pn) given sufficient number of processors; n is the depth of the data-flow and spawning graphs, whereas calculating the min of pn items could take (log pn)time.

3.5. Derivation of a predictive recognizer

The CKY span-control algorithm computes strictly bottom up and enters many values in the recognition table that are eventually useless since they cannot lead to the recognition of S in cell [0,n]. More formally, the requirement on CKY[i,j] is that

```
A ϵ CKY[i,j] iff
      A derives (input i j).
```

There are many ways of introducing some kind of top down prediction. One way is to allow the bottom up method to enter in any cell [i,j] only those nonterminals that "follow" a legitimate parse of (input 0 i). That is, we restrict the entries to obey a second condition:

```
A ϵ CKY-restrict[i,j] iff
      A derives (input i j)
  and S derives xAy such that x is (input 0 i).
```

<u>Define</u> A <u>follows</u> x, for any A ϵ N, x ϵ T*
iff S =>* xAy for some y ϵ V*

Using the index notation, we write:

<u>Define</u> A <u>follows</u> i, for any A ϵ N, $0 \leq i \leq n$,
iff S =>* [0,i]Ay for some y in ϵ V*

We would like to turn this definition into a recursive function <u>Follows</u>, which will be used as a filter in a simple fashion. Follows will be used form an intersection with the already computed cell entries before posting them in the cell.

<u>Algorithm</u> *Schematic version of predictive algorithm*

```
Invoke (member 'S (CKY [0,n]))
Define (CKY [i,j]) =
       case j-i = 1 (Intersect (Follows i) (Lookup (Units (input i j))))
       case j-i > 1
          (Intersect (Follows i)
             (Union for k in (i+1 to j-1)
                (Union for [B,C] in (Cache (CKY [i,k])) cross (Cache (CKY [k,j]))
                   (Lookup (Binaries B C)) )))
```

In the case j−i>1, we can use the distributivity of Union and Intersect functions, to move the Intersect through the two loops, and get:

Algorithm *Schematic version of predictive algorithm*

```
Invoke (member 'S (CKY [0,n]))
Define (CKY [i,j]) =
       case j-i = 1 (Intersect (Follows i) (Lookup (Units (input i j))))
       case j-i > 1
            (Union for k in (i+1 to j-1)
               (Union for [B,C] in (Cache (CKY [i,k])) cross (Cache (CKY [k,j]))
                  (Intersect (Follows i) (Lookup (Binaries B C))) ))
```

Now let us focus on converting the definition of the predicate "A follows x" into a recursive function definition. We introduce case analysis into two parts and treat each part separately. The string x is either the empty string or the string is non-empty.

When the string x is empty, we need to use as a filter all the leftmost symbols appearing in any sentential form. Let us denote by the form "(left* {'S})", a precomputed set containing all nonterminals A such that S =>* Az for some z in V*. This can be defined in terms of an elementary function
(left SetB) = (Union for B in SetB (For [B → AC] in (BP B) collect A)). We will suppress the details of how "left*" is defined in terms of the function "left".

When x is non-empty terminal string, instead of arbitrary derivations, we can consider only leftmost derivations since in S=>*xAy, x is a string of terminals. In a leftmost derivation, A is introduced either by a production of the form B → AC or B → CA. In the first case, using
B → AC, means S=>*xBz=>xACz, and y=Cz. In the second case, using B → CA, means S=>*uBy=>uCAy. Eventually C must derive string v such that x = uv. That is, S=>*uBy=>uCAy=>*uvAy=xAy. Thus, we write,

```
S=>*xAy for non-empty terminal string x iff
either S=>*xBz=>xACz using B → AC in BP,
    or S=>*uBy=>uCAy=>*uvAy = xAy using B → CA in BP.
```

From this we define the predicate follows:

```
A follows x for non-empty string x iff
  either x=uv, B follows u, and C derives v and B → CA in BP,
     or B follows x and B → AC in BP.
```

Converting this to a function (Follows x) that returns a set of nonterminals, we get:

```
(Follows x) =
case |x| = 0  (left* {'S})
case |x| > 0
     (Union (Union for [u,v] in (split x)
                (Union for B in (Follows u)
                   (For [B → CA] in (BP B)
                      collect A if (Derives C v))))
             (Union for B in (Follows x)
                (For [B → AC] in (BP B) collect A)))
```

Notice that this is not an effective definition, since there is an occurrence of (Follows x) in the definition for (Follows x). However, the function has the following recursive structure: (F x) = (Union (G x) (H (F x))). The fixed point for such a recursive structure can be written as (F x) = (Union for i from 0 to Infinity (H^i (G x)))) = (H^* (G x)). In other words,

```
(Follows x) =
case |x| = 0  (left* {'S})
case |x| > 0  (H* (G x))
where (G x) = (Union for [u,v] in (split x)
                 (Union for B in (Follows u)
                    (For [B → CA] in (BP B)
                       collect A if (Derives C v) )))
  and (H SetB) = (Union for B in SetB
                    (For [B → AC] in (BP B) collect A))
```

Both the domain and range of H are finite and hence H^* represents a finitely terminating computation, in view of the fact that H^* will reach its limiting value in a finite number of steps. Typically, in serial computation, a function like (H^* (G x)) would get expressed as

```
Begin Answer := (G x)
      (Repeat until no new element is produced
       Answer := Answer Union (H Answer))
End
```

The definition of H is in fact the function "(left SetB)" introduced in the last page. Thus, H^* is the same as the function left* that we introduced to handle the case j=0.

```
Define (Follows x) =
          case j = 0 (left* {'S})
          case j > 0 (left* (Union for [u,v] in (split x)
                              (Union for B in (Follows u)
                                 (For [B → CA] in (BP B)
                                    collect A if (Derives C v)) )))
```

We now repeat the process we followed earlier in introducing the index notation and simplifying by unfolding the definition of split. This gives us the following definition for Follows.

```
Define (Follows j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for B in (Follows i)
                                (Union for [B -> CA] in (BP B)
                                   collect A if (Derives C [i,j]) ))))
```

Once again, we repeat what by now may be a familiar maneuver, turning the test predicate (Derives C [i,j]) into the loop generator, and using the production as a membership test, in order to isolate a fragment of the algorithm that depends only on the grammar and not on the input string.

```
Define (Follows j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for [B,C] in (Follows i) cross (CKY [i,j])
                                 (For [B -> CA] in (BP B)
                                    collect A))))
```

Now, notice that the expression in the (For ...) is a function of B and C and depends only on the grammar and not on the input string. This can be precomputed, and we denote this by a function (LeadsTo B C). Calling (LeadsTo B C) is done via (Lookup (LeadsTo B C)). We do not carry out this step right now, preparing for further simplifications.

Incorporating this definition into CKY, renaming CKY to CKY-restrict, and introducing the intersection operation as the filter, we obtain the algorithm below.

Algorithm X *A Predictive Recognizer*

```
Define (CKY-restrict [i,j]) =
        case j-i = 1 (Intersect (Units (input i j)) (Follows i))
        case j-i > 1
          (Union for k in (i+1 to j-1)
             (Union for [B,C] in (CKY-restrict [i,k]) cross (CKY-restrict [k,j])
                (Intersect (Binaries B C) (Follows i)) )).
with (Follows j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for [B,C] in (Follows i) cross (CKY [i,j])
                                 (For [B -> CA] in (BP B)
                                    collect A))))
```

The use of (Follows i) allows the left-context of a substring [i,j] to restrict the entries that can be posted in a given cell [i,j]. This form of left-context influence is known experimentally to be very useful in parsing natural language texts. The algorithm of

Ruzzo presented in [16] also uses left-context to restrict the CKY parser, but *generalized to the case of context-free grammars.*

3.5.1. Weakening the predictor

The computation of (Follows j) is quite complex and introduces complex forms of data flow dependencies. More specifically, notice that (i) (Follows j) requires access to all of the results of (Follows i) for i<j, and (ii) (CKY-restrict [i,j] requires access to all of results of (Follows i). These dependencies *effectively* serialize the computation and make it move along the index i. How can we get back the concurrency that we have just lost?

One characteristic of using predictive filters is that we can *weaken* the predictor to gain computational and conceptual simplicity, at the cost of making the main algorithm produce some unneeded extra results in the cells. A weak predictor is one that computes a superset of what is computed by the strong predictor, and thus does not filter as strongly as is possible. The weakest predictor is one that admits any nonterminal in any cell; using this is equivalent to the pure bottom-up algorithm. We started with this version and added a strong predictor that lost great deal of concurrency. Weakening the predictor in this fashion still computes correctly the value for the invocation (member 'S (CKY-restrict [0,n])); it might just store some additional values in various cells of the CKY matrix. We now show an intermediate form of weak predictor, which we will call (Restrictor j).

To arrive at the weakened predictor, called Restrictor, we modify the definition of (Follows i) and replace the occurrence of (Follows i) by the set of nonterminals of the grammar, N.

```
Define (Restrictor j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for [B,C] in N cross (CKY-restrict [i,j])
                                (For [B -> CA] in (BP B)
                                   collect A))))
```

Then we simplify by replacing (BP B) by (BP) and eliminating the loop over N, we get:

```
Define (Restrictor j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for C in (CKY-restrict [i,j])
                                (For [B -> CA] in (BP)
                                   collect A))))
```

Once again, the innermost loop is dependent only on the grammar, not on the input string being parsed, and can be abstracted into a function (Followers C) which can be set up for table look-up.

Algorithm *A Weakened Predictor*

```
Define (Restrictor j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for C in (CKY-restrict [i,j])
                                (Lookup (Followers C))
with (Followers C) =
     (For [B -> CA] in (BP) collect A)
```

Both occurrences of the call to (Follows i) in Algorithm X are replaced by (Restrictor i) to arrive at Algorithm XI.

Algorithm XI *A Weakly Predictive Recognizer*

```
Define (CKY-restrict [i,j]) =
        case j-i = 1 (Intersect (Lookup (Units (input i j)))
                                (Cache (Restrictor i)))
        case j-i > 1
          (Union for k in (i+1 to j-1)
             (Union for [B,C] in (Cache (CKY-restrict [i,k]))
                             cross (Cache (CKY-restrict [k,j]))
                (Intersect (Lookup (Binaries B C))
                           (Cache (Restrictor i))) ))

with (Restrictor j) =
        case j = 0 (left* {'S})
        case j > 0 (left* (Union for i in (1 to j-1)
                             (Union for C in (Cache (CKY-restrict [i,j]))
                                (Lookup (Followers C))))
with (Followers C) =
     (For [B -> CA] in (BP) collect A)
```

The above weakening was done by decoupling the call to (Follows i) within (Follows j). We can also pursue an alternate weakening of the predictor, decoupling the call to (CKY-restrict [i,j] within (Follows j). This would yield a restrictor function that is recursive and depends only on the grammar, hence could be precomputed. Only actual experiments can tell us which of these predictors are practically useful. The second form of Restrictor has the following simplified definition. Within CKY-restrict, the call to Restrictor2 will appear as (Lookup (Restrictor2 i)).

```
Define (Restrictor2 j) =
       case j = 0 (left* 'S)
       case j > 0 (left* (Union for i in (1 to j-1)
                           (Union for B in (Restrictor2 i)
                             (Lookup (Followers2 B))
with (Followers2 B) =
     (For [B -> CA] in (BP B) collect A)
```

3.5.2. Relaxed synchronization of predictor and recognizer

The data flow dependencies between Follows/Restrictor and CKY-restrict do not permit the extent of parallelism we witnessed in the CKY-span-control. The synchronization requirement for CKY-restrict moves along successive diagonals of the recognition matrix, whereas that for Follows/Restrictor moves along its columns. The combination of these two destroy a substantial extent of the parallelism available. We can, however, *relax the strict synchronization* to produce an asynchronous parallel algorithm that is non-deterministic. We may allow lax synchronization in the access to CKY-restrict in calculation of Follows/Restrictor and in the access to Follows/Restrictor in the calculation of CKY-restrict. The effect of such a relaxed synchronization is acceptable; *the algorithm is still functionally accurate* [24]. That means that Follows/Restrictor may have not produced its final value and CKY-restrict wants to access it (and vice versa).

We introduce a new form for cache, *(Cache-Initial (f . args) initial-value)*. If the cache cell for (f . args) indicates that it is being computed or has not yet been invoked, the initial-value is used by this call. Otherwise, the stored cache value is used. This caching construct is *nondeterministic* because the value it result it produces depends on the physical concurrency and scheduling characteristics of the implementation. We replace the cached call to Restrictor from CKY-restrict by a Cache-Initial call; and also the cached call to CKY-restrict from Restrictor by a Cache-Initial call. The cache we use for access to Follows/Restrictor supplies all of the set N if Follows/Restrictor has not produced its final value. Similarly the cache for CKY-restrict will supply all of N to Follows/Restrictor if its final value is not yet available.

Algorithm XI modified *A Weakly Predictive Asynchronous Recognizer*

```
Define (CKY-restrict [i,j]) =
       case j-i = 1 (Intersect (Lookup (Units (input i j)))
                               (Cache-Initial (Restrictor i) N))
       case j-i > 1
         (Union for k in (i+1 to j-1)
           (Union for [B,C] in (Cache (CKY-restrict [i,k]))
                               cross (Cache (CKY-restrict [k,j]))
              (Intersect (Lookup (Binaries B C))
                         (Cache-Initial (Restrictor i) N)) ))

with (Restrictor j) =
       case j = 0 (left* {'S})
       case j > 0 (left* (Union for i in (1 to j-1)
                           (Union for C in (Cache-Initial (CKY-restrict [i,j]) N)
                              (Lookup (Followers C))))
with (Followers C) =
     (For [B -> CA] in (BP) collect A)
```

A variation on the above can be obtained by using the form (Cache-Initial (Restrictor i) (Lookup (Restrictor2 i))) instead of (Cache-Initial (Restrictor i) N). Since Restrictor2 is precomputed it may supply somewhat filtered set in comparison to using all of N.

3.5.3. Adaptive scheduling of tasks

We can also assign priorities to tasks generated by Follows/Restrictor & CKY-restrict, that depend both upon the task type and the parameter values for the invocation. We can control the relative rates of progress for the bottom-up calculation and for the imposing of restriction. One could experiment with the possibility that this type of priority assignment might be done automatically and adaptively. If the computation of CKY-restrict moves faster than that of Follows/Restrictor, it will have to deal with larger sets and may gets slowed down, allowing Follows/Restrictor to catch up. On the other hand, if Follows/Restrictor moves faster it will slow down waiting for delivery of values from CKY-restrict. Thus, there may be an interesting form of negative feedback governing the functioning of these two functions.

3.6. Derivation of a version of Earley's algorithm

The algorithm published by Earley [14] is most interesting for its unique character. It works on unrestricted context-free grammars and runs in serial time proportional to the n^3 on inputs of length n. What makes it interesting is that when the grammar used is regular, it works in linear time. Earley comments on the fact that this happens automatically and that the user need not identify the grammar to be regular.

In fact, the class of grammars for which it runs in linear time is wider than this. Also, in the general case, the algorithm does NOT require any additional conditions such as the absence of non-producing symbols or empty-productions, as many other algorithms do.

We now outline the serial Earley recognition algorithm, restricted for CNF grammars. Restricting it to CNF grammars gives us continuity to our development and also allows us to conveniently put in correspondence our algorithm with Earley's. We do not foresee any difficulty in giving the same treatment for the more general case of context-free grammars.

Algorithm Serial Earley

This algorithm is modified from Earley's paper to be limited to CNF grammars and uses no lookahead. The productions in the original grammar are numbered in some arbitrary order from 1 to $|UP|+|BP|$. A dummy production numbered 0 is added to get the process started. This dummy production has as its right-hand-side the only symbol 'S. Its left-hand-side is an unused new nonterminal symbol and has no significance.

```
   Define (Earley G Input) =
```

The algorithm computes successively a sequence of n+1 state-sets, numbered from 0 to n. Each state-set indicates results of parsing the input string up to position i. Each element of a state-set is a 3-tuple [p,f,j],
[p = number identifying the Production,
f = number of symbols of thr RHS recognized so far,
j = starting position of the input string].

A tuple is said to be final *if all the RHS symbols have been recognized; otherwise, it is* non-final. *Thus in the case of CNF grammars, a tuple [p,f,j] is final for a unit production p if f=1, and is final for a binary production p if f=2. Initially, the state-sets are empty sets.*

```
   Body Let $S_j$ be empty for $0 \leq i \leq n+1$

        Add [0,0,0] to $S_0$.
```

Here the first element of the tuple is production 0, the dummy production. The second element is also 0 indicating that no symbol has been recognized so far. The third element is also 0 indicating that the input string is to be scanned from position 0. Since the dummy production has S for its only right hand side, this initialization sets up the state-set to seek to recognize S.

```
        For i from 0 step 1 until n do
        Process the elements of state $S_i$,
        performing one of the following operations on each tuple [p,f,j].
```

If the next symbol of the production to recognize is a terminal and it matches the input string, advance the pointer into the production.

```
(1) Scanner: If the tuple is non-final and the next symbol of
   p is a terminal (i.e. f=0 and p is in UP), and if the i+1th
   input character matches the j+1th symbol of p, then
   add to S_{i+1} the tuple [p,f,j+1].
```

If the next symbol is a nonterminal, for each production alternative q for it, enter a new tuple pointing to 0, and scan the input from the same place.

```
(2) Predictor: If the tuple is non-final and the next symbol A of
   p is a nonterminal (i.e. f=0 or 1 and p is in BP), then for each
   q = [A -> BC] or q = [A -> x]
   add to S_i the tuple [q,0,i].
```

If the pointer is at the end of a production, take each tuple that predicted this and advance its pointer.

```
(3) Completer: If the tuple is final
    (i.e. f=' and p in UP or f=2 and p in BP)
    then for each [q,l,g] in S_j in which the next symbol of q
    (i.e. the l+1th symbol) is equal to the left-side of p,
    add to S_i the tuple [q,l+1,g].

If S_{i+1} is empty, return Rejection.
If i=n and [0,2,0] is in S_{n+1}, then return Acceptance.
```

The algorithm as presented needs detailed explanation and proof, which is lacking in Earley's paper, but we are told they are available in his thesis. What is the key idea? We attempt to distill this in our own development of Earley's algorithm, one that runs in parallel.

We fall back to Algorithm II and pursue a different line of development that leads us towards Earley's algorithm adapted for CNF grammars.

<u>Algorithm II</u>

```
Define (Derives A x) =
       case |x| = 1 (member [A -> x] (UP A))
       case |x| > 1 (For Some [[A -> BC],[y,z]] in (BP A) cross (split x)
                               (and (Derives B y) (Derives C z) ))
```

We attempt to eliminate the need for (split x) and the corresponding loop over [y,z]. We convert the above definition so that, given A and x, it attempts to derive from A some prefix substring of x, and to return the split in the string corresponding to the prefix. Let us call the new function Absorb. Absorb accepts a nonterminal A and a substring x of the the input string, and returns a set of pairs [y,z] meaning that nonterminal A can derive string y, and string z is left over, so that x = yz. To do recognition, we invoke Absorb passing it 'S and the Input string; we test whether the entire input string is recognized to be 'S by asking whether the pair [Input,""] is a member of the result. Thus if the entire input string is derivable from 'S and the empty string "" is left over, we have a string that is accepted by the grammar.

```
[y,z] ∈ (Absorb A x) implies (Derives A y)

[x,""] ∈ (Absorb A x) implies (Derives A x)
```

In doing the conversion, we have to observe that since Absorb seeks to derive only a prefix of the input string, the case analysis of Algorithm II needs to be modified so that the first case is for $|x| \geq 1$ and the second case is for $|x| > 1$. However, to keep the cases disjoint, we adopt the case break down of $|x| = 1$ and $|x| > 1$, and repeat the treatment of unit productions, UP, so that it appears in both the first and second case.

Algorithm XII

```
Invoke (member [Input,""] (Absorb 'S Input))
Define (Absorb A x) =    ; returns a set of string pairs
       case |x| = 1
         (if (member [A -> (first x)] (UP A)) then {[(first x),(rest x)]} else {})
       case |x| > 1
         (Union (if (member [A -> (first x)] (UP A))
                    then {[(first x) , (rest x)]} else {})
                (Union for [A -> BC] in (BP A)
                    (Union for [y,z] in (Absorb B x)
                        (For [p,q] in (Absorb C z)
                            collect [y||p , q]))))
```

Once again, we introduce index pairs to denote substrings. Thus, the form of the invocation will be (member [[0,n],[n,n]] (Absorb 'S [0,n])). This yields the following.

Algorithm XIII

```
Invoke (member [[0,n],[n,n]] (Absorb 'S [0,n]))
Define (Absorb A [i,n]) = ; returns a set of pairs of index pairs
       case n-i = 1
         (if (member [A -> (input i i+1)] (UP A))
          then {[[i,i+1],[i+1,n]]} else {})
       case n-i > 1
         (Union (if (member [A -> (input i i+1)] (UP A))
                    then {[[i,i+1],[i+1,n]]} else {})
                (Union for [A -> BC] in (BP A)
                    (Union for [[i,j],[j,n]] in (Absorb B [i,n])
                        (For [[j,k],[k,n] in (Absorb C [j,n])
                             collect [[i,k],[k,n]])))))
```

The second element of the second argument to Absorb will always be n, since it never changes. We can clearly omit this from the parameter structure and and simplify We then get the following algorithm, renamed to Endpositions.

Algorithm XIV

```
Invoke (member n (Endpositions 'S 0))
Define (Endpositions A i) = ; returns a set of end positions
        case n-i = 1
          (if (member [A -> (input i i+1)] (UP A)) then {i+1} else {})
        case n-i > 1
          (Union (if (member [A -> (input i i+1)] (UP A)) then {i+1} else {})
              (Union for [A -> BC] in (BP A)
                  (Union for j in (Endpositions B i)
                      (Endpositions C j))))
```

Let us abstract out the first part of the result to be a function Endpositions1 which will be suitable for caching and then we also cache the two calls to Endpositions.

Algorithm XV

```
Invoke (member n (Endpositions 'S 0))
Define (Endpositions A i) = ; returns a set of end positions
        case n-i = 1 (Cache (Endpositions1 A i))
        case n-i > 1
          (Union (Cache (Endpositions1 A i))
              (Union for [A -> BC] in (BP A)
                  (Union for j in (Cache (Endpositions B i))
                      (Cache (Endpositions C j)))))
with (Endpositions1 A i) =
      (if (member [A -> (input i i+1)] (UP A)) then {i+1} else {})
```

A straightforward interpretation of the algorithm XIV might get trapped in a loop if the grammar contains left-recursion, whereas with the use of cache, algorithm XV might hang up waiting for cache delivery. A circular cache-request will never deliver. A simple case of this is when a production is of the form [A -> AC]. Scherlis [29] describes a transformation that turns this into a finitely terminating computation, called the "Finite Closure Transformation". Given the domain of the function is finite, the only way a recursive algorithm can loop forever is because the function is being called with exactly the same arguments. The least-fixed-point semantics of such a function can be arrived at by replacing all redundant calls to the function by the Identity of the operation that combines the results. In our case the results are combined by the Union operation. Thus, trapping redundant calls and making these return {} will give us a finitely terminating computation that determines the least fixed point.

In our situation this transformation is effected simply by using a different caching function, CacheDefault. Its behavior is described simply as follows: (CacheDefault (f x y) DefaultValue) retrieves and returns the result from the cache if (f x y) has been requested and completed. If there is no pending request for it, it will create a request entry and initiate computation of (f x y). In case there is a pending request a more subtle action is required, because of the possibility of concurrent calls. If the request is a circular one, that is, the one who initiated the computation is a spawning ancestor of the current request, then immediately a value of DefaultValue is returned to the current request. Otherwise, the current request is not a circular one, and the current request waits for the other request to complete[3].

Algorithm XVI A parallel version of Earley algorithm

```
Invoke (member n (Endpositions 'S 0))
Define (Endpositions A i) = ; returns a set of end positions
       case n-i = 1 (Cache (Endpositions1 A i))
       case n-i > 1 (Union (Cache (Endpositions1 A i))
                         (Union for [A -> BC] in (BP A)
                            (Union for j in (CacheDefault (Endpositions B i) {})
                               (CacheDefault (Endpositions C j) {}))))
with (Endpositions1 A i) =
     (if (member [A -> (input i i+1)] (UP A)) then {i+1} else {})
```

To see the correspondence to the Earley algorithm we must explain the three operations that Earley uses: Scanner, Completer and Predictor, and also explain the differences in the way we express the control structure and the way he does it. Earley structured his computation to be based on producing a sequence of n+1 state-sets numbered from S_0 to S_n, each state set containing a tuple[4]. A tuple [p,f,i] entered in state set S_j means that the production p has been partially recognized, starting at the substring of the input beginning at i and ending at j, where f (the

[3]The implementation of CacheDefault is quite intricate; we have to be careful to distinguish between the two kinds of returns in addition to the two kinds of pending requests. One kind of cache return is the return from a circular call with the default value; the other kind of return is a normal return. One kind of cache request is a circular request; the other is not. There are many issues involved in the design of Cache memory. Specifically, we have not yet addressed the issue of cache retention period, and thus the issue of when to clear the cache.

[4]We are simplifying the Earley algorithm by omitting the look-ahead feature; the tuple in the original algorithm contains a fourth element, which is the look-ahead string. The original algorithm takes an additional parameter which indicates the size of the look-ahead string to be computed; our algorithm corresponds to zero length look-ahead. The algorithm complexity is unaffected by variations in this parameter.

pointer) indicates the number of symbols in the production's right side have been successfully recognized. In a CNF grammar, f can be 0, 1 or 2 if p is in BP and f can be 0 or 1 if p is in UP.

```
Case p is in UP and f=0:
        Apply scanner
Case p is in UP and f=1:
        Apply ccmpleter
Case p is in BP and f=0 or 1:
        Apply predictor
Case p is in BP and f=2:
        Apply completer
```

The function Endpositions1 and the expression (if ... then ... else ...) defining it constitutes the Scanner. It reads one terminal symbol in the input string and moves the pointer past the nonterminal that needed to be recognized. The part of the program that translates a call to (Endpositions A i) to (Endpositions B i) for some production [A —> BC] is the Predictor. The continuation which would then call (Endpositions C j) based on the value of j returned by (Endpositions B i) is also part of the Predictor. The return of a function call continues the computation that was in effect before the function call; this operation corresponds to the Completer operation of Earley.

I trust that with moderate effort the reader is able to see how the above algorithm applies the proper operators to the proper cases. Since the Unions can be all be invoked in parallel, the only necessary serialization is the data-flow dependency that requires the arrival of values j from the calls to (Endpositions B i). This is, again, more concurrency than the "obvious" concurrency one might have captured in directly translating Earley algorithm into a parallel form[5].

4. Conclusion

We have illustrated the systematic derivation of parallel algorithms for context-free recognition. We have derived versions of the two most well-known general context-free recognition algorithms *in addition to deriving several new ones*. Similar treatments of sorting and other algorithms have been previously presented in the

[5]There is a possible trap in anyone's first attempt in parallelizing Earley's algorithm. One would be tempted to process each tuple in the state-set S_i in parallel, retaining the serial processing of the sets S_0 through S_n. However, this cannot be done so readily, since the action of the predictor in state S_i is to add more tuples to the same state-set S_i.

literature [7, 12, 32]. Our work is distinguished in two respects. We have attempted a somewhat more complex problem than sorting algorithms and we have derived several new algorithms in addition to reproducing known ones. Deak [13] has presented a derivation of the CKY algorithm; we invite the reader to compare the level of perspicuity in the derivations that Deak gives with the ones presented here. We suspect that the complexity of Deak's derivation stems from the introduction of side-effect causing construction very early in the development, and also, specifically, the choice of assignment to a variable (rather than caching as is in our case) further complicates matters. Amarel [3] has given a general formulation of the problem of syntactic analysis as a theorem-proving problem in a reduction system. He shows sets of reduction rules that do bottom-up, top-down, left-right as well as right-left derivations. He shows that by altering the attention focusing method systematically, that is varying the priorities on rule application, his formulation spans a wide range of possible algorithms. Included in his formulation are some new as well as old algorithms. However, we were not able to detect in his range of algorithms the equivalent of the Earley algorithm; this algorithm requires a specific type of abstraction (abstraction over one of the parameters, thus producing the function Absorb) that is not included in the formulation given by Amarel. Since Earley published his results two years after Amarel's report was made available, this is perhaps not surprising. *Our paper makes a contribution in the area of parsing by showing the close affinity between the CKY and Earley algorithms as well as by displaying a new mixed top-down and bottom-up algorithm and a dynamic programming algorithm.*

The major thrust of the paper, however, is the demonstration of *semi-applicative programming*, that is, using an applicative programming language combined with program transformations, program annotation (precedence control and caching) and adaptive scheduling. Context-free parsing is just one area of study; we are in the process of dealing with a wider range of common problems that arise in Artificial Intelligence programming. We hope we have given the reader a sample of semi-applicative programming by focusing on one specific case. The use of precedence control and the several forms of caching in combination with program transformations are useful ways of controlling the behavior of applicative programs. At a high level, *Tuning for performance need not be a machine-oriented activity; and it can be done in a safe and reliable manner preserving the functionality of the program.* This might be considered a contribution to programming methodology.

Postscript

I am indebted to Douglas Smith of Kestrel Institute, who read an earlier draft of this paper and brought to my attention two papers by H. Partsch [27, 28]. Partsch has demonstrated transformational developments for both the CKY and the Earley algorithms executable on parallel machines. Indirectly, I have learned that there is also related work by Scherlis [30] and Jones [19] but strictly for the serial case. In the final form of this manuscript I hope to relate my work to theirs. The development of the predictive top–down recognizer and its weakened and asynchronous versions, the dynamic programming solution appear, in any case, to be novel results for this paper.

Acknowledgement

It is my pleasure to thank Andy Haas for the help he has offered always with enthusiasm. He has also pushed me to search for simplicity for which I am grateful. I also thank my several colleagues at BBN Laboratories who have provided me their generous help and support.

References

[1] H. Abelson et al.
The revised report on Scheme or An uncommon Lisp.
Technical Report AI Memo 848, MIT, August, 1985.

[2] D. Allen and staff.
A Lisp for the Butterfly parallel processor.
Technical Report, BBN Laboratories, (in progress).

[3] S. Amarel.
Problem-solving procedures for efficient syntactic analysis.
Technical Report Scientific Report 1, RCA Laboratories, Princeton, NJ, May, 1968.

[4] G.R. Andrews and F.B. Schneider.
Concepts and notations for concurrent programming.
Technical Report TR 82-520, Cornell University, New York, September, 1982.

[5] J. Backus.
Can programming be liberated from the Von Neumann style?
Communications of the ACM 21(8):613-641, July, 1982.

[6] R. Balzer.
A 15 year perspective on automatic programming.
IEEE Transaction on Software Engineering SE-11(11):1257-68, November, 1985.

[7] D.R. Barstow.
Knowledge-Based Program Construction.
Elsevier North-Holland, New York, 1979.

[8] BBN Staff.
Butterfly parallel processor overview.
Technical Report, BBN Laboratories, 10 Moulton Street, Cambridge MA 02238, June, 1985.

[9] W. Bibel.
Syntax-directed, semantics-supported program synthesis.
Artificial Intelligence 14:243-261, 1980.

[10] R.S. Bird.
Tabulation techniques for recursive programs.
Computing Surveys 12(4):403-417, December, 1980.

[11] F. W. Burton.
Annotations to control parallelism and reduction order in the distributed evaluation of functional programs.
ACM Transactions on Programming Languages and Systems 6(2):159-174, April, 1984.

[12] J. Darlington.
A synthesis of several sort programs.
Acta Informatica 11(1):1-30, 1978.

[13] E. Deak.
A transformational derivation of a parsing algorithm in a high level language.
IEEE Trans. on Software Engineering SE-7(1):23-31, January, 1981.
Language used is a variant of SETL.

[14] J. Earley.
An efficient context-free parsing algorithm.
Communications of the ACM 13(2):94-102, February, 1970.

[15] R.P. Gabriel and J. McCarthy.
Queue based multi-processing Lisp.
Technical Report STAN-CS-84-1007, Stanford University, June, 1984.

[16] S.L. Graham, M. Harrison, W.L. Ruzzo.
An Improved Context-Free Recognizer.
ACM Transactions on Programming Languages and Systems 2(3):415-462, July, 1980.

[17] R. Halstead Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages , October, 1985.

[18] J.E. Hopcroft and J.D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison-Wesley, Reading, MA, 1979.
See Chapter 4.

[19] C.B. Jones.
Software Development: A rigorous approach.
Prentice-Hall, Englewood Cliffs, NJ, 1980.

[20] T. Kasami and K. Torii.
A syntax-analysis procedure for unambiguous context-free grammars.
Journal of the ACM 16(3):423-431, July, 1969.

[21] R.M. Keller and M. Ronan Sleep.
Applicative caching.
In *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 131-140. Association for Computing Machinery, October, 1981.

[22] D.F. Kibler and J. Conery.
Parallelism in AI programs.
In *Proc. 9th IJCAI*, pages 53-56. Morgan Kaufman Publishers, 1985.

[23] R. Kowalski.
Logic for Problem Solving.
Elsevier North-Holland, New York, 1979.

[24] V.R. Lesser and D.D. Corkill.
Functionally accurate cooperative distributed systems.
In *Proc. 1st Int'l Conference on Cybernetics and Society*, pages 346-353. IEEE, October, 1979.

[25] J. McCarthy et al.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, MA, 1963.

[26] H. Partsch and R. Steinbruggen.
Program transformation systems.
Computing Surveys 15(3):199–236, 1983.

[27] H. Partsch.
Structuring Transformational Developments: A case study based on Earley's recognizer.
Science of Computer Programming 4:17–44, April, 1984.

[28] H. Partsch.
Transformational Derivation of Parsing Algorithms Executable on Parallel Architectures.
In U. Ammann (editor), *Programmiersprachen und Programm-Entwicklung*, pages 47–57. Informatik-Fachberichte 77, Springer-Verlag, 1984.

[29] J. Reif and W. Scherlis.
Deriving efficient graph algorithms.
Technical Report, CMU, 1982.

[30] W.L. Scherlis.
Expression procedures and program derivation.
Technical Report STAN-CS-80-818, Stanford University, 1980.

[31] E. Shapiro.
A subset of Concurrent Prolog and its interpreter.
Technical Report TR-003, ICOT – Institute for New Generation Computing, Tokyo, Japan, February, 1983.

[32] D.R. Smith.
Top-down synthesis of divide-and-conquer algorithms.
Artificial Intelligence 27(1):43–96, September, 1985.

[33] N.S. Sridharan.
A semi-applicative language for artificial intelligence programming.
Technical Report, BBN Laboratories, November, 1984.

[34] P.C. Treleaven, D.R. Brownbridge and R.P. Hopkins.
Data-Driven and demand-driven computing architecture.
Computing Surveys 14(1):93–143, March, 1982.

Official Distribution List

Contract N00014-85-C-0079

	Copies
Scientific Officer	1
Head, Information Sciences Division	
Office of Naval Research	
800 North Quincy Street	
Arlington, VA 22217-5000	
Attn: Dr. Alan L. Meyrowitz	
Mr. Frank Skieber	1
Defense Contract Administration	
Services Region - Boston	
495 Summer Street	
Boston, MA 02210-2184	
Director, Naval Research Laboratory	1
Attn: Code 2627	
Washington, DC 20375	
Defense Technical Information Center	12
Bldg. 5	
Cameron Station	
Alexandria, VA 22314	